CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:1813–1835
Published online 23 October 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1507

Modeling of tsunami waves and atmospheric swirling flows with graphics processing unit (GPU) and radial basis functions (RBF)

Jessica Schmidt[1], Cécile Piret[2], Nan Zhang[3], Benjamin J. Kadlec[4],
David A. Yuen[5,*,†], Yingchun Liu[5], Grady Barrett Wright[6]
and Erik O. D. Sevre[5]

[1] *College of Saint Scholastica, Duluth, MN 55811, U.S.A.*
[2] *National Center for Atmospheric Research, Boulder, CO 80305, U.S.A.*
[3] *CREST, Medical School, University of Minnesota, MN 55455, U.S.A.*
[4] *Department of Computer Science, University of Colorado, CO 80309, U.S.A.*
[5] *Minnesota Supercomputing Institute, University of Minnesota, MN 55455, U.S.A.*
[6] *Department of Mathematics, Boise State University, ID 83725, U.S.A.*

SUMMARY

The faster growth curves in the speed of graphics processing units (GPUs) relative to CPUs have spawned a new area of development in computational technology. There is much potential in utilizing GPUs for solving evolutionary partial differential equations and producing the attendant visualization. We are concerned with modeling tsunami waves, where computational time is of extreme essence in broadcasting warnings. We employed an NVIDIA board on a MacPro to test the efficacy of the GPU on the set of shallow-water equations, and compared the relative speeds between CPU and GPU for two types of spatial discretization based on second-order finite differences and radial basis functions (RBFs). We found that the GPU produced a speedup by a factor of 8 in favor of the finite difference method and a factor of 7 for the RBF scheme. We also studied the atmospheric dynamics problem of swirling flows over a spherical surface and found a speedup of 5.3 by the GPU. The time steps employed for the RBF method are larger than those used in finite differences, because of the fewer number of nodal points needed by RBF. Thus, RBF acting in concert with GPU would hold great promise for tsunami modeling because of the spectacular reduction in the computational time. Copyright © 2009 John Wiley & Sons, Ltd.

Received 8 December 2008; Accepted 24 July 2009

KEY WORDS: RBF; GPU; tsunami

[*] Correspondence to: David A. Yuen, Minnesota Supercomputing Institute, University of Minnesota, MN 55455, U.S.A.
[†] E-mail: daveyuen@gmail.com

Contract/grant sponsor: NSF; contract/grant number: ATM-0620100
Contract/grant sponsor: Advanced Studies Program at NCAR

1. INTRODUCTION

Natural catastrophic disasters, like tsunamis, commonly strike with little warning. For most people, tsunamis are underrated as major hazards (e.g. [1]). People wrongly believed that they occur infrequently and only along some distant coast. Tsunamis are usually caused by earthquakes [2]. Seismic signals usually can give some margin of warning, since the speed of tsunami waves travels at about 1/30 of the speed of seismic waves. Still there is not much time, between 1 h and a few hours for distant earthquakes and much less, if you happen to be unluckily situated in the near-field region. Figure 1 shows an artist's impression of the tsunami caused by March, 1964 Alaskan earthquake. The power associated with tsunami waves may be imagined by Figure 1. Therefore, it is important to have codes that are fast to respond to the onset of tsunami waves. It is desirable to have codes that can deliver the output in the course of a few minutes. With the introduction of graphics processing unit (GPU) as a commodity item into the computer graphics market, it is timely to adopt this new technology for solving partial differential equations (PDEs) describing the propagation of tsunami waves. This may have important consequences in the warning strategy, if a factor of around ten can be achieved on a single CPU.

There has already been some work done using GPUs for solving equations involving fluid dynamics by the group at E.T.H. [3,4] dealing with complicated physics, such as bubble formation and wave breaking. Computational efforts in molecular dynamics [5], astrophysics [6] seismic wave propagation in 3D [7], and other geoscience areas [8], using CUDA, have also been carried out on GPUs.

In the next section we will give the mathematical equations used in the modeling. This will be followed by an introduction to the concepts of GPU and CUDA, a recently developed software, which allow one to translate readily existing codes written in C language to programs capable of being run on a GPU. We then give a brief introduction to RBFs and the software, Jacket, which can translate the RBF code written in MATLAB to CUDA form capable of running seamlessly on the GPU.

Radial basis functions (RBFs) (e.g. [9]) are a novel method to solve PDEs. They represent a gridless approach [10] and require fewer grid points for solving the PDEs because of its high accuracy. We will discuss their potential use in the shallow-water equations together with their implementation on a GPU. We then give the results on the comparison in computational times of CPU versus GPU for both the linear shallow-water equations and the swirling flow problem in atmospheric flows. In the last section we summarize our findings and give some perspectives for future work.

2. TSUNAMI EQUATIONS

We will now give a brief summary of how tsunamis are generated by earthquakes and how tsunami waves propagate across the sea. While ordinary storm waves break and dissipate most of the energy in a surf zone, tsunami waves break at the shoreline. They lose little energy as they approach a coast and can run up to heights of an order of magnitude greater than storm waves. The reader is kindly referred to [11–15] for a more thorough review of the physics and classification of tsunami waves and the numerical techniques employed in the modeling.

DOI: 10.1002/cpe

Figure 1. This image was illustrated by Pierre Mion for Popular Science in 1971, in response to an earthquake that caused a tsunami near Prince William Sound, Alaska in 1964. The train pictured was carried 50 m by the humongous waves [1].

In brief, tsunami waves, which typically have periods spanning from 100 s to 2000 s, are generated by earthquakes by means of transfer of large-scale elastic deformation associated with earthquake rupturing process to increase in the potential energy in the water column of the ocean. Most of the time, the initial tsunami amplitude waves are very similar to the static, coseismic vertical displacement produced by the earthquake. In tsunami modeling [16], one commonly calculates the elastic coseismic vertical displacement field from elastic dislocation theory (e.g. [17]) with a single Volterra (uniform slip) dislocation. Geist and Dmowska [13] showed the fundamental importance that the distributed time-dependent slip have on tsunami wave generation. We note that because variations in slip are not accounted for, these simple dislocation models may underestimate the coseismic vertical displacement that dictates the initial tsunami amplitude. It is only recently (e.g. [18]) that horizontal displacements are deemed to be important in tsunami modeling.

After the excitation due to the initial seafloor displacement, the tsunami waves propagate outward from the earthquake source region and follow the inviscid shallow-water wave equation, since the tsunami wavelength (around hundreds of km) is usually much greater than the ocean depths. For water depths greater than about 100–400 m [19,20], we can approximate the shallow-water wave equations by the linear long-wave equations

$$\frac{\partial z}{\partial t}+\frac{\partial M}{\partial x}+\frac{\partial N}{\partial y}=0 \tag{1}$$

$$\frac{\partial M}{\partial t}+gD\frac{\partial z}{\partial x}=0 \tag{2}$$

$$\frac{\partial N}{\partial t}+gD\frac{\partial z}{\partial y}=0 \tag{3}$$

where the first equation (1) represents the conservation of mass and the next two equations (2), (3) govern the conservation of momentum. The instantaneous height of the ocean is given by $z(x, y, t)$, a small perturbative quantity. The horizontal coordinates of the ocean are given by x and y and t is the elapsed time, M and N are the discharge mass fluxes in the horizontal plane along the x and y axes, g is the gravitational acceleration, $h(x, y)$ is the undisturbed depth of the ocean, and the total water depth is

$$D(x, y, t)=h(x, y)+z(x, y, t) \tag{4}$$

It is important to emphasize here that the real advantage of the shallow-water equation is that z is a small quantity and hence a perturbation variable, which allows it to be computed accurately.

The three variables of interest in the shallow-water equations are $z(x, y, t)$, the instantaneous height of the seafloor, and the two horizontal velocity components $u(x, y, t)$ and $v(x, y, t)$. We will employ the height z as the principal variable in the visualization. The wave motions will be portrayed by the movements of the crests and troughs in the wave height z, which are advected horizontally by u and v.

A thorough discussion of the limitations of the shallow-water equations in both the linear and nonlinear limits, as well as 3-D waves, can be found in a recent lucid contribution by Kervella *et al.* [21]. Shallow-water, long wavelength equations are commonly solved by using second-order accurate finite difference techniques (e.g. [19]). As a rule of thumb, there should be 30 grid points covering the wavelength of a tsunami wave. For a tsunami with a wave period of 5 min, this criterion requires a grid size of 100 m and 500 m, where the depth of the water exceeds, respectively, 10 m and 250 m. Accurate depth information is more important than the width of the grid in modeling tsunami behavior close to the coast. We note that the shallow-water equations describing tsunamis are best employed for distances of about a few hundred kilometers from the source of earthquake excitation. Otherwise, the full set of 3-D Navier–Stokes equations should be brought to bear [15,22], especially in light of the recent suggestion about the importance of horizontal movement in tsunami excitation [18].

DOI: 10.1002/cpe

3. COMPUTING TSUNAMIS ON GPUs

In 2004, a humongous tsunami struck Sumatra, killing approximately 230 000 people. Never before has a tsunami been known to be this deadly. Perhaps, if a method or computational hardware tools had been available that could model tsunamis quickly and accurately after the onslaught of an earthquake, many people may still be alive today. The average depth of the ocean is roughly 5000 m. If a tsunami-causing earthquake were to strike, a tsunami wave would travel close to 800 km/h. Although the wave slows, as it approaches the shoreline, it continues to move quite quickly. The depth of the ocean near the shore is typically around 100 m, thus, the wave continues to move at over 110 km/h [23]. Tsunamis are difficult to detect as they usually look like most other ocean waves. The only warning is the earthquake. On average, a tsunami vertically displaces between 12 and 23 inches of water. Therefore, the best way to accurately predict where a tsunami will strike and how large it grows is through numerical simulations. Thus, the quicker a simulation produces data, the quicker a tsunami warning could be issued. Figure 2 shows the generation and propagation of a tsunami by an earthquake in a subducting region.

Within the last decade, commodity GPU specialized for rendering of 2D and 3D scenes has seen an explosive growth in the processing power compared with their general purpose counterpart, the

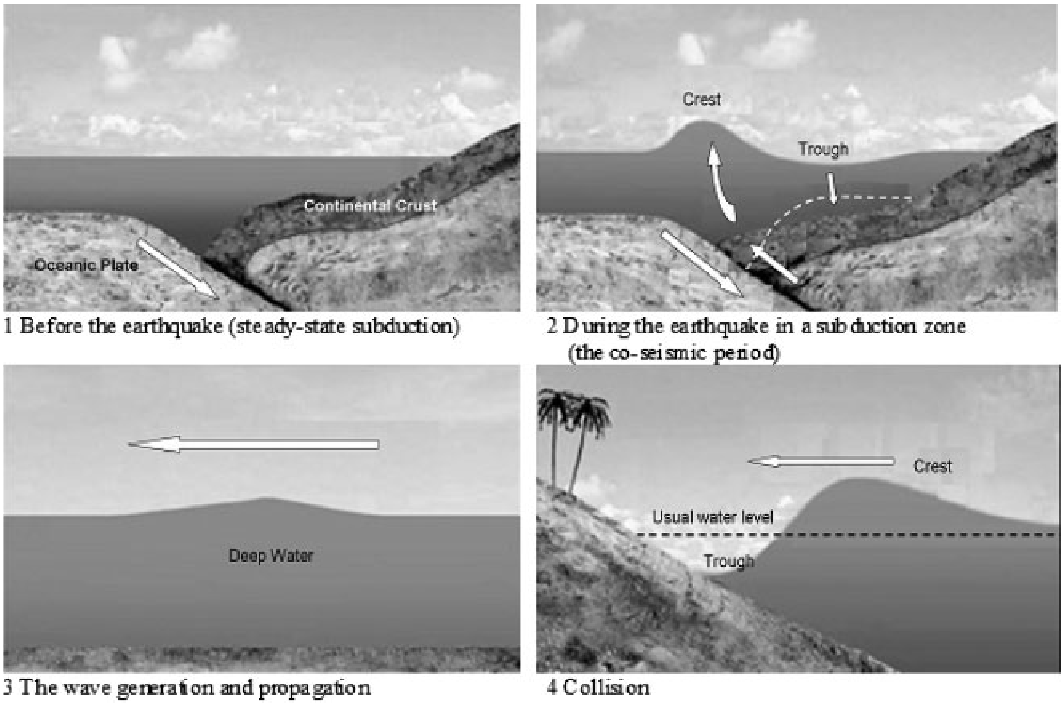
Continental Crust
Oceanic Plate
1 Before the earthquake (steady-state subduction)
Crest
Trough
2 During the earthquake in a subduction zone
(the co-seismic period)
Deep Water
3 The wave generation and propagation
Crest
Usual water level
Trough
4 Collision

Figure 2. This demonstrates how the tsunami is generated and how it propagates through the ocean. ([24], Adapted from universe-review.ca/F09-earth.htm).

DOI: 10.1002/cpe

CPU. Currently, capable of near teraflop speed and sporting gigabytes of on-board memory, GPUs have indeed transformed from accessory video game hardware to potentially useful computational co-processors. However, a GPU can also be used to compute complex mathematical operations, thereby lifting the burden off the CPU and allowing it to dedicate its resources to other tasks. More recently, the programming research community has come up with programming models that would map well onto GPUs. NVIDIA's CUDA, which was introduced in 2007, treats the GPU as a SIMD processor and allows for general purpose computing. CUDA marked both a redesign of the hardware, plus the addition of a new software layer to accommodate general purpose computing. When our research group saw the potential speedup of implementing simulations with a GPU armed with CUDA, we decided to investigate whether we could adapt our computational problems for a GPU. We looked at modeling tsunamis through two different methods: the finite difference method and RBFs to solve our PDEs. The GPU we used to implement the finite difference method tsunami simulation was an NVIDIA GeForce 8800 and an NVIDIA GeForce 8600M GT to implement the RBF simulation. The use of these GPUs permitted us to achieve a speedup over running the simulations on the CPU alone.

The GPU has multiple types of memory buffers available on it, and when used correctly, they can further speedup a simulation. There are significant advantages to reading from texture memory as compared with the global GPU memory. Therefore, our simulations use both the texture and linear memory since texture is necessary to experience the full benefits of the GPU architecture. Textures act as low-latency caches that provide high bandwidth for reading and processing data. Thus, we are able to read data on the GPU very quickly, since it is essentially a memory cache. Textures also provide linear interpolation of voxels through texture filtering that allows for the ease of calculations done at sub-voxel precision. Data access using textures also provides automatic handling for out of bounds addressing conditions, such that sloppy programming can forgiven by automatically clamping to the extents of a volume or wrapping to the next valid voxel.

Coalesced memory access refers to accessing consecutive global GPU memory locations by a group of CUDA threads (in the same warp) and creates the best opportunity to maximize the memory bandwidth. Unfortunately, many applications cannot be mapped to coalesced reads and therefore an expensive increase in the latency results in significantly less than optimal bandwidth. Fortunately, CUDA provides the opportunity to map global memory to a texture that allows data to be entered in a local onchip cache with significantly lower latency. In order to be optimal, the texture cache still requires locality in data fetches, but it provides significantly more flexibility especially when using multi-dimensional textures for 2D and 3D reads. Therefore, memory reads using texture fetching can significantly increase the memory bandwidth, as long as there is some locality in the fetches. For our purposes, we are always making local texture fetches from memory as our computation requires access only to neighboring voxels in the 2D data. In practice, texture memory can be accessed in 1–2 cycles resulting in bandwidths around 70 GB/s (86.40 GB/s theoretical max) as compared with global (non-coalesce) memory reads that require a significant 400–600 cycle latency that results in a poor bandwidth of 3.5 GB/s. However, it needs to be stressed that these numbers vary greatly depending on exact memory access patterns and how often the texture cache needs to be updated.

There is a tradeoff though, for our techniques, since using texture memory requires the allocation of an additional volume in the global memory. As the texture cache is not guaranteed to remain coherent (clean) when global memory writes occur in the same function call, we need to ensure

DOI: 10.1002/cpe

that no data is written to the global memory pointed to by a texture-mapped address. Therefore, we cannot read and write from the same volume during a single kernel call and must write to a temporary output volume that can then be copied to the texture-mapped memory after the completion of the kernel call. Writing to global memory during a kernel call is only relevant to texture-mapped linear memory (kernels can never write to CUDA arrays), but care must be taken since the undefined data is returned by a texture fetch when the cache loses coherency (i.e. when dirty).

However, since it is very time consuming to write to texture memory, we found that the optimal process for our particular simulation was to use linear memory much of the time. Although linear memory takes a considerable amount of time to read, it can easily be written to. Therefore, we would copy the data from the linear memory into texture, so that we could read the data quickly, while also being able to quickly update the values in linear memory. A problem we ran into with texture memory is that many of our calculations referenced multiple arrays, each of whose index referenced a different position. Therefore, we were unable to use texture memory as much as we would have liked to. Currently, we are working on resolving this issue in order to eliminate reading data from linear memory, and reading only from texture.

If possible, eventually we would also like to visualize the tsunami while the simulation is running. However, visualizing the tsunami on the GPU is only possible if the entire data set can fit on the GPU. Otherwise, we will need to continue writing the data back to the CPU for the visualizations. Currently, we are returning to the CPU to write our data to a file every 60 time steps. After the simulation has finished its run, we then take those files and import them into visualization software called Amira [24]. Therefore, if our data is small enough, we may be able to eliminate the constant movement between the CPU and the GPU, which may allow us to attain greater speedup. Even if the visualization on the GPU is not possible, ideally, we would like to compute and visualize the projected path of the tsunami in faster than real time, thus enabling a tsunami warning to be issued to the people in the vicinity of the tsunami path before the wave arrives.

4. CUDA AND TSUNAMI COMPUTATION

As discussed above, we have elected to use CUDA, which can be downloaded for free from the NVIDIA web site along with its compiler. There is a large user community in CUDA, including a few dozen universities which use it in classes. This programming development greatly facilitates the GPU to be used as a data-parallel supercomputer with no need to write functions within the restrictions of a graphics API. NVIDIA designed CUDA for their G8x series of graphics cards, which includes the GeForce 8 Series, the Tesla series, and some Quadro cards. We will describe only the basic details of the CUDA programming model, but for further details, we advise the reader to refer to the CUDA Programming Guide [25]. Before GPU programming languages became widely accessible, the only way to program a GPU was through assembly language. Assembly language is very difficult to implement; therefore, not many people attempted GPU programming. Recently, GPU programming has become more accessible through the development of a variety of languages, such as RapidMind from Waterloo, Brook from Stanford and then CUDA. The CUDA programming interface is an extension of the C programming language and therefore provides a relatively easy learning curve for writing programs that will be executed on the device. Within the

DOI: 10.1002/cpe

CUDA language, the CPU is commonly called the host and the GPU is the device. As already mentioned, CUDA programs follow a SIMD paradigm where a single instruction is executed many times, but independently on different data and by different threads. The resulting program, which we call a kernel, needs to be written as an isolated function that can be executed many times on any block of the data volume.

There are some limitations to GPU programming with CUDA. For example, CUDA only works on certain graphics cards, all of which are developed by NVIDIA. These cards include the GeForce 8000 series, along with a few selected Teslas and Quadros. Therefore, it may be difficult to find a computer with the ability to run CUDA. Additionally, CUDA only supports 32-bit floating point precision on the GPU. Although it does recognize the double variable type, when a double is cast to the GPU, it is reduced to a float. Currently, a new language is being developed by AMD called Brooks+, which would support 64-bit precision; however, currently no such language exists, but these problems will ameliorate.

Another limitation to GPU programming is the bottleneck caused by the latency and bandwidth between the CPU and GPU, since it is easier to copy data from the CPU to the GPU rather than vice versa. To understand this more fully, we will describe how the CUDA program works. First, all of the variables need to be set up for both the host and device. When allocating linear memory on the GPU, we use the command `cudaMalloc()`. Additionally, during this time we also set up the texture memory locations as well. Next, we populate the data on the host before calling the kernel. In order to execute our kernel on the device, we must copy the data to the GPU. To perform this operation, we use `cudaMemcpy()` to copy the data from the CPU to the GPU. Finally, once the data has been copied to the device, the kernel can be executed. The kernel in effect runs in a parallel manner on the GPU processors as each thread executes the kernel simultaneously, thus decreasing the elapsed time the CPU would have needed to run it in a sequential manner. When the kernel reaches the end, the program returns to the host. However, in order to work with the data computed on the GPU, we need to use the `cudaMemcpy()` command to copy the data from the GPU to the CPU. Now, the program may continue its execution. Figure 3 illustrates this process. Finally, right before the program ends, we want to be sure to use the `cudaFree(device_variable)` to ensure that the device's memory locations are cleared so that it may perform its other tasks optimally. Therefore, as the number of data transfers between the CPU and GPU increase, the less effective the GPU becomes.

A major advantage of the CUDA architecture over prior GPU programming environments is the availability of DRAM memory addressing, which allows for both scatter and gather memory operations, essentially allowing a GPU to read and write memory in the same way as a CPU. CUDA also provides a parallel onchip shared memory that allows threads to share data with each other and read and write very quickly. This shared memory feature circumvents many expensive calls to DRAM and reduces the bottleneck of DRAM memory bandwidth.

Overall, we found CUDA to be the best language to fulfill our needs. In order to accomplish the task of GPU programming, we found it beneficial and the most useful to first port the finite-difference tsunami simulation from FORTRAN 77 to C. Although it was possible to keep the simulation in FORTRAN and call upon CUDA kernels, we believed it to be easier if the entire simulation was written using the same language. However, later when we were faced with the task of putting MATLAB code on the GPU, we felt a different option through the Jacket software which was more constructive. Jacket will be elaborated on later in this paper.

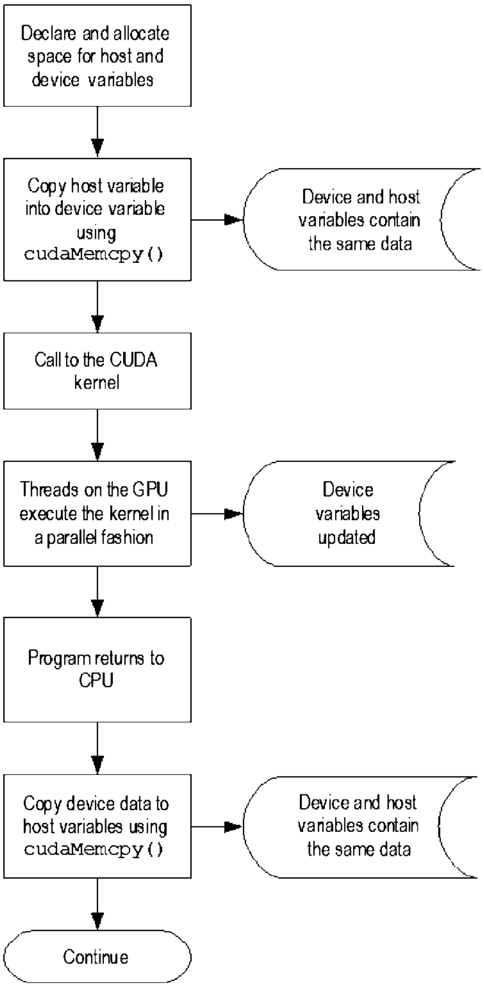
Declare and allocate space for host and device variables
Copy host variable into device variable using cudaMemcpy()
Device and host variables contain the same data
Call to the CUDA kernel
Threads on the GPU execute the kernel in a parallel fashion
Device variables updated
Program returns to CPU
Copy device data to host variables using cudaMemcpy()
Device and host variables contain the same data
Continue

Figure 3. Flow chart portraying how a typical CUDA program operates and the potential bottlenecking involved with the data transfer.

5. SAMPLE CUDA CODE FOR GPU

The next two figures demonstrate how a CUDA kernel can be written and how it is called from the main program. This example doubles every value stored in the array, using addition. Figure 4 represents the kernel that will be executed inside the GPU, while Figure 5 is the rest of the program that is run on the CPU.

DOI: 10.1002/cpe

```
__global__ void addArray(int *a, int size)
{
     int i = blockIdx.x * blockDim.x + threadIdx.x;
     if (i < size)
          a[i] = a[i] + a[i];
}
```

Figure 4. This is an example of a CUDA kernel. It displays how arrays can be added together on the GPU.

```
main()
{
   int size = 10;                 // the size of the arrays
   int *a_h, *a_d;                // creates two pointer arrays, one
                                  // for each the host and device
   size_t N = size * sizeof(int);
   a_h = (int *)malloc(N);        // allocates space on CPU for array
   cudaMalloc((void**)&a_d, N);   // allocates space on GPU for array

   // initialize the host array to desired values
   ...

   // set the thread and grid size
   ...

   // copies the contents of the host array into the device array
   cudaMemcpy(a_d, a_h, N, cudaMemcpyHostToDevice);
   // call to the CUDA kernel
   addArray<<<grid, thread>>>(a_d, size);
   // copies the contents of the device array into the host array
   cudaMemcpy(a_h, a_d, sizeof(int)*size, cudaMemcpyDeviceToHost);

   for (int i = 0; i < size; i++)
   {
      printf("%d\n", a[i]);       // prints results
   }
   free(a_h); cudaFree(a_d);      // frees the host  and device memory
}
```

Figure 5. This section is taken from the main part of the program. It shows how the arrays are set up for the GPU, and how to call the CUDA kernel. Moreover, it displays how the data is copied back to the CPU after being computed on the GPU.

6. A DESCRIPTION OF RBF METHODOLOGY

6.1. Introduction

Rolland Hardy (1971) introduced the RBF methodology with what he called the Hardy multiquadric (MQ) method. The method originally came about in a cartography problem, where scattered bivariate data needed to be interpolated to represent topography and produce contours. The common interpolation methods of the time (e.g. Fourier, polynomials, bicubic splines, etc.) were not guaranteed to produce a non-singular system with any set of distinct scattered nodes. It can be shown, in fact, that when the basis terms of an interpolation method are independent from the nodes to be interpolated, there is an infinite amount of node sets leading to a singular system. Hardy's method bypassed this issue. It was innovative in that his method represented the interpolant as a linear

DOI: 10.1002/cpe

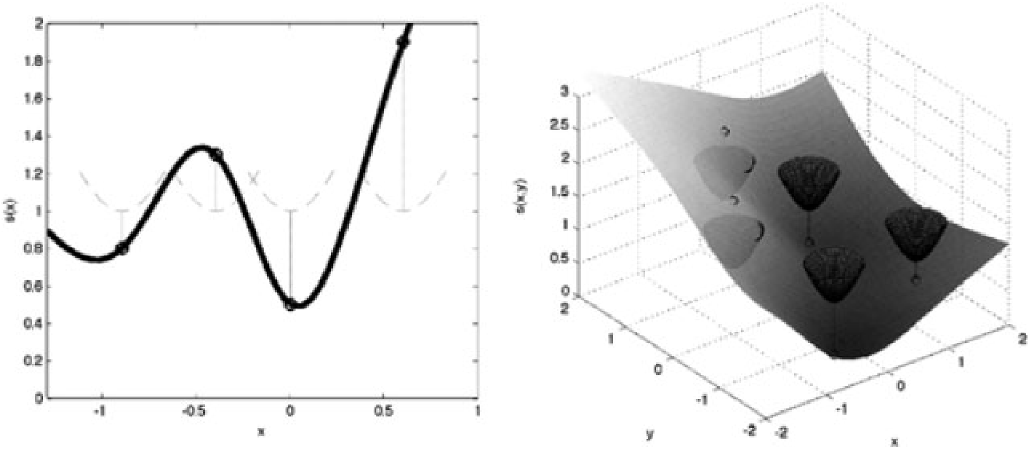

Figure 6. The RBF method consists in centering a radial function at each node location and imposing that the interpolant take the node's associated function value.

combination of one basis function (originally the MQ function, Figure 6), centered at each node location, making the basis terms dependent on the nodes to be interpolated. Furthermore, it was shown that the basis terms of Hardy's method produced an interpolation system that was unconditionally non-singular. Although orthogonality of the basis terms was lost, a wellposedness was achieved for any set of scattered nodes and in any dimension. Throughout the years, more such 'radial functions' were used than the original MQ with which Hardy introduced the method (Figure 7). All radial functions have the particular property to only depend on the Euclidean distance from their center, making them radially symmetric. The name of the method was therefore generalized as the RBF method. It was not until the 1990s, with Ed Kansa, that RBFs were used to solve PDEs for the first time [26]. Although the method is young and still relatively unknown, it offers great prospects for modeling in geophysical fluid dynamics.

6.2. RBF representation

Given the data values f_i at the scattered node locations $\underline{x}_i$, $i = 1, 2, \ldots, n$ in d dimensions, an RBF interpolant takes the form

$$s(\underline{x}) = \sum_{i=1}^{n} \lambda_i \phi(\|\underline{x} - \underline{x}_i\|) \tag{5}$$

where $\|\cdot\|$ denotes the Euclidean $L2$-norm.

We obtain the expansion coefficients λ_i by solving a linear system $A\underline{\lambda} = \underline{f}$, imposing the interpolation conditions $s(\underline{x}_i) = f_i$. The system takes the form

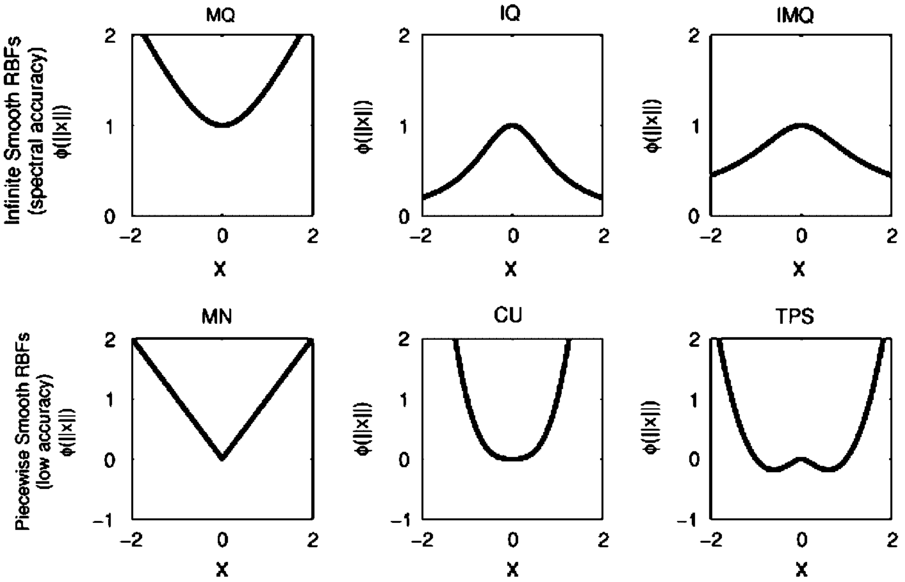

Figure 7. Commonly used radial basis functions. The piecewise smooth RBFs only give rise to low accuracy, while the infinitely smooth RBFs provide spectral accuracy. MN is shown in the case of $k=1$, that is, $\phi(r)=r$.

$$\begin{bmatrix} \phi(\|x_1-x_1\|) & \phi(\|x_1-x_2\|) & \dots & \phi(\|x_1-x_n\|) \\ \phi(\|x_2-x_1\|) & \phi(\|x_2-x_2\|) & & \phi(\|x_2-x_n\|) \\ \vdots & & & \vdots \\ \phi(\|x_n-x_1\|) & \phi(\|x_n-x_2\|) & \dots & \phi(\|x_n-x_n\|) \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \tag{6}$$

It is sometimes necessary to append a low-order polynomial term to the RBF interpolant in order to guarantee the non-singularity of the collocation matrix. More on this subject and on RBFs in general can be found in [10].

There are two kinds of radial functions: the piecewise smooth and the infinitely smooth radial functions. The piecewise smooth radial functions have a jump in one of their derivatives, which limits them to yielding only an algebraic accuracy. The infinitely smooth radial functions, on the other hand, offer spectral accuracy. They have a shape parameter, ε, which controls how steep they are. The closer this parameter is to 0, the flatter the radial function becomes. Table I contains some of the most commonly used piecewise and infinitely smooth radial functions $\phi(r)$.

It is interesting to note this heuristic reasoning behind the spectral accuracy of the infinitely smooth radial functions. In 1D, the cubic radial function $\phi(r)=r^3$ has a jump in its 3rd derivative, making its interpolant $O(h^4)$ accurate (h is inversely proportional to the number of node points, N. It can be thought of as the typical node distance, since no grid is required.) The quintic radial function, $\phi(r)=r^5$, has a jump on its 5th derivative and leads to an $O(h^6)$ accurate interpolant. In general,

DOI: 10.1002/cpe

Table I. Definitions of some of the most common radial functions.

Name of RBF	Abbreviation	$\phi(r), r \geq 0$	Smoothness
Multiquadric	MQ	$\sqrt{1+(\varepsilon r)^2}$	Infinitely smooth
Inverse multiquadric	IMQ	$\frac{1}{\sqrt{1+(\varepsilon r)^2}}$	
Inverse quadratic	IQ	$\frac{1}{1+(\varepsilon r)^2}$	
Generalized multiquadric	GMQ	$(1+(\varepsilon r)^2)^\beta$	
Gaussian	GA	$e^{-(\varepsilon r)^2}$	
Thin plate spline	TPS	$r^2 \log(r)$	Piecewise smooth
Linear	LN	r	
Cubic	CU	r^3	
Monomial	MN	r^{2k-1}	

the MN radial function $\phi(r)=r^{2k-1}$ has a jump in its $2k-1$st derivative and its interpolant will be $O(h^{2k})$ accurate. Thus, the smoothness of the radial function is the key factor behind the accuracy of its interpolant. The piecewise continuous radial functions therefore converge algebraically toward the interpolated function, as we increase the number of node points. We note here that a radial function could not take the form $\phi(r)=r^{2k}$ since it could interpolate a maximum of $2k+1$ nodes (in 1-D), due to the fact that the resulting interpolant reduces to a polynomial of degree $2k$. On the other hand, a radial function that is infinitely continuously differentiable (and not of polynomial form) will produce a spectrally accurate interpolant, which converges as $O(\mathrm{e}^{-const/h})$ toward the interpolated function, if no counterpart to the Runge phenomenon enters [27]. The Gaussian RBF is an exception to the rule, as it converges as $O(\mathrm{e}^{-const/h^2})$, that is, ‘super-spectrally’ [28]. This rule holds on 1-D equispaced grids, but equivalent results seem to hold also in higher dimensions when using scattered nodes.

The accuracy of the infinitely smooth radial functions also depends on their shape parameter and can be improved by changing the flatness of the radial function. The limit of $\varepsilon \to 0$ has become very interesting in that respect. The range of small ε used to be inaccessible because of the ill-conditioning that it caused. Since the introduction of the Contour–Padé method developed by Fornberg and Wright [29], this obstacle commonly known as ‘the uncertainty principle’ was lifted and it was finally possible to explore the features of the small ε RBFs. Recently, the RBF-QR algorithm was introduced by Fornberg and Piret [9]. Similar to the Contour–Padé method, it allows to compute the RBF interpolant in the low ε regime. However, unlike the Contour–Padé method, the RBF-QR method is not limited to work for only small numbers of nodes.

7. SOFTWARE USED IN MATLAB FOR TRANSLATING RBF INTO CUDA

Although MATLAB provides an API to interface with C code, and in essence with CUDA through MEX files [30], we decided to use software developed by AccelerEyes called Jacket to run the RBF simulation in conjunction with a GPU. However, we decided to use software developed by AccelerEyes called Jacket. By using Jacket, we are able to access the GPU without leaving the

```
A = eye(5);            % creates a 5x5 identity matrix

A = gsingle (A);       % casts A from the CPU to the GPU

A = A * 5;             % multiplies matrix A by a scalar on the GPU

A = double (A);        % casts A from the GPU back to the CPU
```

Figure 8. Demonstrates how a MATLAB program can take advantage of the GPU by using the software Jacket.

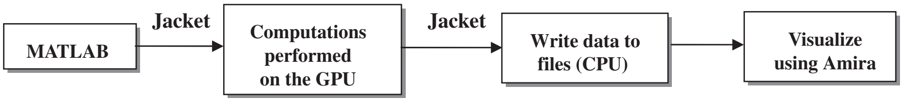

Figure 9. The configuration of the current MATLAB simulation.

MATLAB environment. Jacket is an engine that runs CUDA in the background, eliminating the need for the user to know any GPU programming languages. Instead of writing CUDA kernels, one just needs to tell the MATLAB environment when and what should be transferred to the GPU and then when to copy it back to the CPU [31]. See Figure 8 for an example on how to implement MATLAB code on the GPU by using Jacket.

As Jacket wraps the MATLAB language into a form that is compatible with the GPU, the commands that would have otherwise been written in *C* are eliminated. However, the CUDA drivers and toolkit must be installed on the computer before Jacket can be used. Additionally, once MATLAB has opened, a path needs to be added to the Jacket directory so that MATLAB knows where it can access the files that would allow it to cast the data onto the GPU. Moreover, we are visualizing images on the GPU through MATLAB because, thus far, Jacket only supports OpenGL in Linux. There are plans to expand Jacket so that it can support OpenGL in other environments as well. When this becomes available, we shall be able to visualize the RBF within the native MATLAB environment on any computer that supports CUDA, thereby eliminating the step of writing data back to the CPU. Thus, by using RBFs to model tsunamis in conjunction with the GPU would enable us to visualize the tsunami faster than it is propagating through the water, allowing a tsunami warning to be issued readily. Figure 9 demonstrates how our simulation in MATLAB currently works. Further information about Jacket can be found in the user guide distributed by AccelerEyes [32].

8. COMPARISON OF GPU AND CPU RESULTS

8.1. Linear tsunami waves

After implementing both the finite difference method and the RBF shallow-water simulations on the GPU using CUDA, we received significant speedup in the simulation's run times. The finite difference method simulation was run upon an NVIDIA 8800 GPU. This simulation has a

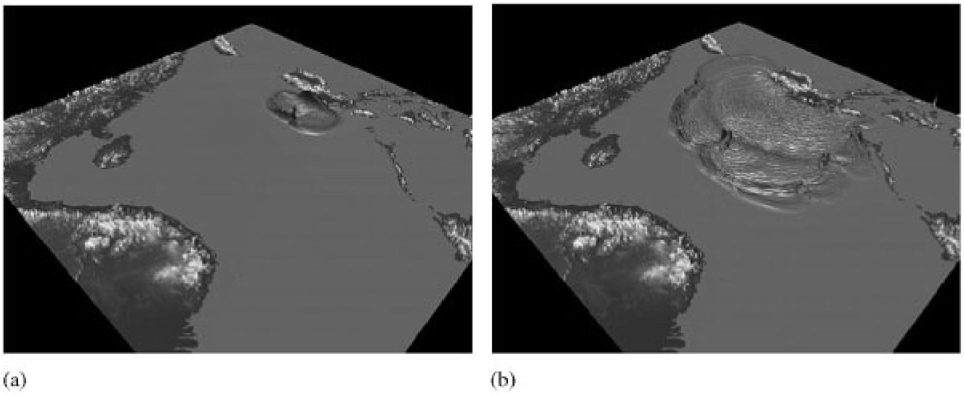

Figure 10. The tsunami as visualized in Amira. The first image (a), shows the tsunami early in its propagation, while the second one (b), illustrates the tsunami later in the simulation. Visually, there is no difference in running the simulation on the GPU than with the CPU.

two-dimensional grid size of 601 × 601 and contains 21 600 time steps, where each time step represents one second. Originally, the simulation was run upon an Opteron-based system in its original form of FORTRAN 77 and it took over 4 h to complete its run. However, running the same simulation in conjunction with the GPU took approximately half an hour. Thus, the simulation was about eight times faster than when using the CPU alone. However, even with the GPU, we are still outputting a file every 60 time steps so that we can visualize the tsunami in Amira. Figure 10 shows a visualization of the data with Amira. If we could eliminate writing data to a file, but rather produce images in real time, the speedup could be increased, since it takes a considerable amount of time to copy data back to the CPU [33]. Therefore, implementing a visualization interface using OpenGL would allow us to visualize the data, as it is being created.

Moreover, we also implemented the RBF simulation on an NVIDIA 8600 GPU using the software Jacket. Comparing the simulation that ran strictly on the CPU of a MacBook Pro to the simulation that implemented a GPU, the speedup we received was about seven times faster. This simulation contained 400 time steps and a grid size of 30 × 30. Overall, we found that running an RBF simulation in conjunction with the GPU would produce the speediest results, thereby allowing the shortest time in issuing a tsunami warning.

8.2. Swirling flows

Another area of interest is using RBFs to model atmospheric simulations. This includes swirling flow problems such as solid body rotation, which are found in weather models. Meteorologists resort to these types of models in order to predict the weather; however, these simulations can take a very long time to run. Therefore, it is very difficult to predict the weather far into the future. One example that we looked at dealt with solid body rotation; specifically, we looked at how the height field of a cosine bell as modeled by Flyer and Wright [34] would travel around the earth. In order

DOI: 10.1002/cpe

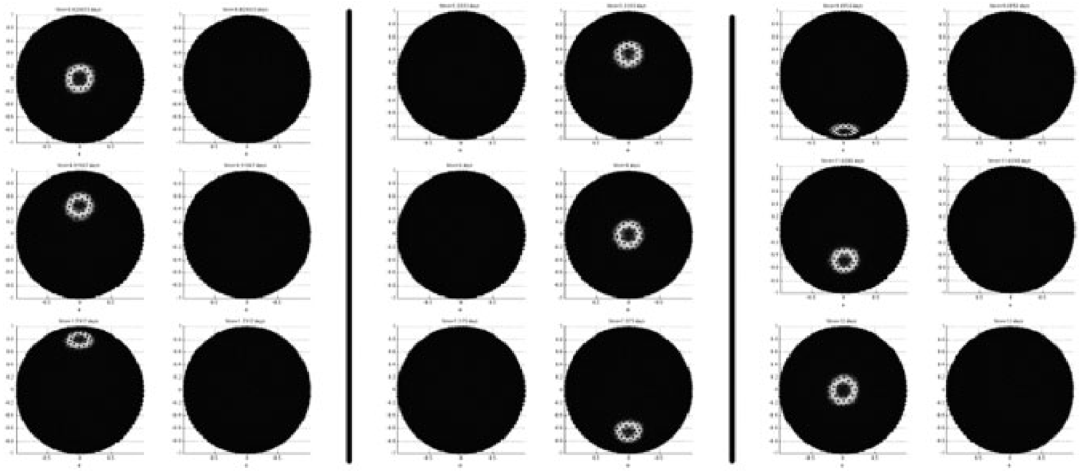

Figure 11. This is an image of the cosine bell traveling around the sphere. The pairs of the sphere images show both sides of the sphere simultaneously, meaning, one side is the front of the sphere, and the other side is the back. Initially, the bell begins at the equator and then moves northward, where it eventually travels to the other side of the sphere. After 12 days, the bell finally returns to its starting location.

to look at this phenomena, the following equations were solved in spherical coordinates:

$$\frac{\partial h}{\partial t}+\frac{u}{a\cos\theta}\frac{\partial h}{\partial\lambda}+\frac{v}{a}\frac{\partial h}{\partial\theta}=0 \tag{7}$$

$$u=u_0(\cos\theta\cos\alpha+\sin\theta\cos\lambda\sin\alpha) \tag{8}$$

$$v=-u_0\sin\lambda\sin\alpha \tag{9}$$

The first equation (7) models the advection of the height field, while the next two equations (8), (9) demonstrate the movement of the wind, with a representing the radius of the earth and u_0 being the speed of the rotation of the earth. The angles θ and λ represent the latitude and longitude, respectively, and α is the angle relative to the pole of the standard longitude–latitude grid. Following Flyer and Wright's model, one complete revolution is made every 12 days. Initially, the cosine bell is centered at the equator and begins by following a northward path around the globe. Figure 11 shows the cosine bell traveling around a spherical object, such as the world (Figure 12).

As RBFs are able to employ much larger time steps compared with other methods, this simulation is able to complete 12 days in a very short amount of time. Visually, there is no difference between the results of the currently accepted methods and RBFs. Since, RBFs can model this phenomenon quickly and accurately, we decided to look into placing the simulation on the GPU by using Jacket since the simulation was written in MATLAB. After placing the most computationally extensive parts of the simulation on the GPU, we calculated a speedup of about 5.3 times faster than running the simulation on the CPU alone. Therefore, if RBFs are used to model weather simulations, which are subsequently placed upon a GPU, forecasters would be able to look at the weather much further into the future, without losing the precision they currently have (Figure 13).

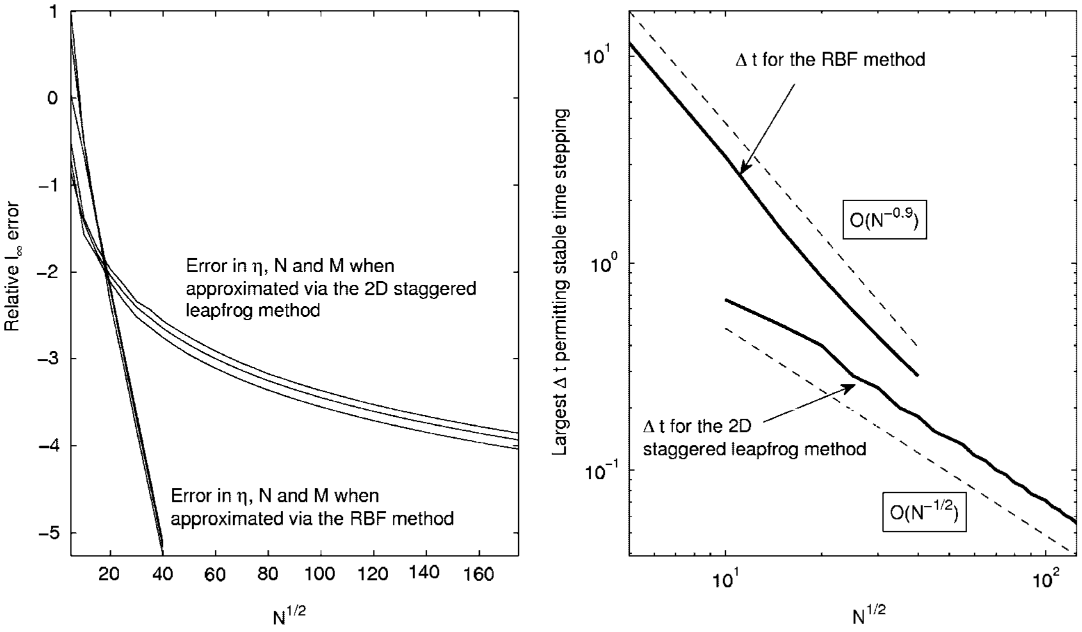

Figure 12. The left figure shows the log of the relative l_∞ error in computing η, M and N via both the RBF method (we note the spectral accuracy—in fact, $error_{RBF} \sim O(\mathrm{e}^{-0.35N^{1/2}})$) and the staggered leapfrog method (second-order accurate in time and in space—$error_{SL} \sim O(N)$). The right figure shows a plot of the largest Δt allowing for stable time stepping for both methods. The dashed lines follow the asymptotic leading behaviors of the Δt curves: $\Delta t_{RBF} \sim N^{-9/10}$ and $\Delta t_{SL} \sim N^{-1/2}$, which is consistent with the CFL condition in time stepping.

8.3. Comparison in physical time steps between finite differences and RBFs

8.3.1. Tsunami governing equations

We have employed Equations (1)–(3), in their simplest form, which are the linearized long-wave equations without bottom friction in two-dimensional propagation. They take the form of the coupled PDE system

$$\eta_t + M_x + N_y = 0 \tag{10}$$

$$M_t + D(x, y)\eta_x = 0 \tag{11}$$

$$N_t + D(x, y)\eta_y = 0 \tag{12}$$

where η is the water depth and M and N are the discharge fluxes in the x and y directions, respectively. The function $D(x, y)$ in Equations (11) and (12) incorporates the bathymetry and is illustrated in Figure 14. We assume for sake of simplicity periodic boundary conditions.

DOI: 10.1002/cpe

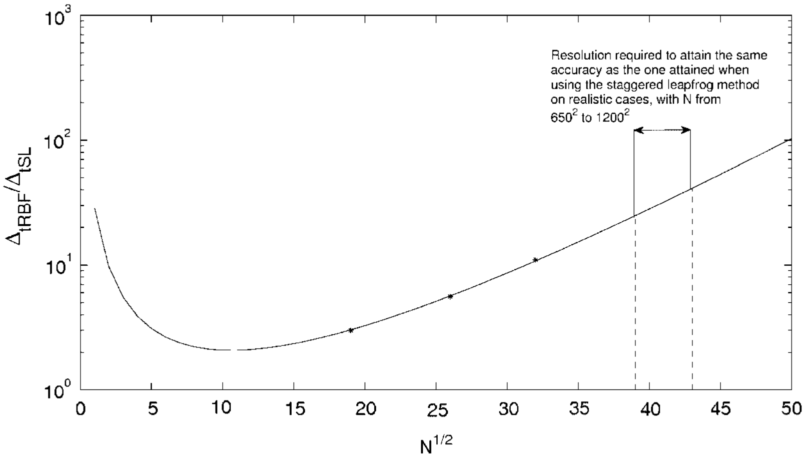

Figure 13. This plot shows the ratio $\frac{\Delta t_{RBF}}{\Delta t_{SL}}$ between the time steps needed for the two methods to attain the same accuracy. This ratio is calculated via the asymptotic behaviors detailed in the legend of Figure 12. It is a good fit to the data gathered in Table II, noted by *.

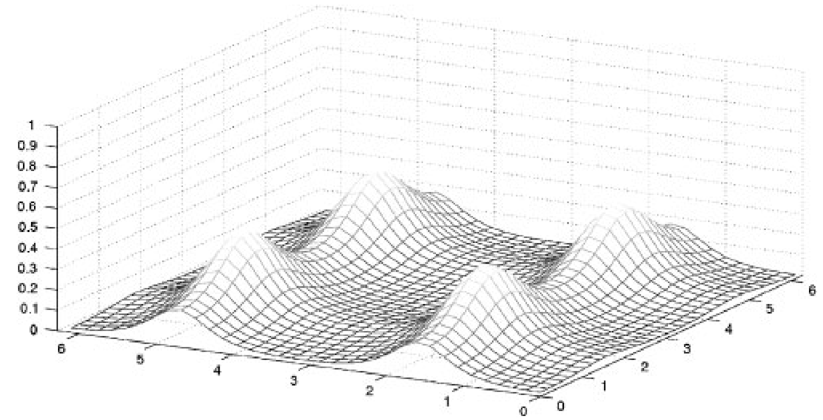

Figure 14. The topography enters the equation in the function $D(x, y)$ of Equations (11) and (12). For this toy problem, we chose $D(x, y) = \frac{15}{(2+\cos^2 x)^2(3+6\cos^2 y)^2}$.

We use the method of lines (e.g. [35,36]) adapted to RBFs. It consists in discretizing the PDE in space using RBFs, and solving the resultant system of ordinary differential equations in time with an ODE integrator, such as the 4th order Runge–Kutta. Discretizing a differential operator L in terms of RBFs can be done as follows. Let us define

$$u(\underline{x}) = \sum_{i=1}^{n} \lambda_i \phi(\|\underline{x} - \underline{x}_i\|) \tag{13}$$

Applying an operator L on both sides gives

$$Ls(\underline{x}) = \sum_{i=1}^{n} \lambda_i L\phi(\|\underline{x} - \underline{x}_i\|) \tag{14}$$

where $L\phi(r)$ can be analytically determined. We can evaluate Equations (13) and (14) at all the nodes and obtain in matrix form, respectively, $\underline{u} = A\underline{\lambda}$ and $\underline{v} = B\underline{\lambda}$. Matrix A is unconditionally non-singular, thus we can eliminate $\underline{\lambda}$ and obtain $\underline{v} = BA^{-1}\underline{u}$. The newly formed matrix $E = BA^{-1}$ is the PDE's differentiation matrix. In our case, we let E_1, E_2, E_3, and E_4 be the differentiation matrices, respectively, corresponding to the operators $\partial/\partial x$, $\partial/\partial y$, $D(x, y)\partial/\partial x$, and $D(x, y)\partial/\partial y$. Let

$$F = -\begin{pmatrix} 0 & E_1 & E_2 \\ E_3 & 0 & 0 \\ E_4 & 0 & 0 \end{pmatrix} \tag{15}$$

Thus, F is the matrix discretizing the spatial operator in the system of Equations (10)–(12). The remaining system of ODEs in time is the following:

$$\underline{u}_t = F\underline{u} \tag{16}$$

where $\underline{u} = (\eta, M^{\mathrm{T}}, N^{\mathrm{T}})^{\mathrm{T}}$. It is customary to use an ODE method such as RK4 to solve Equation (16), so long as the differentiation matrix' spectrum fits in its stability region. The periodic boundary conditions in space can be integrated in the RBF definition itself by making use of the fact that the domain is 2D doubly periodic (topologically equivalent to the nodes laying on the surface of a torus). The Euclidean distance between two points on the unit circle is $2|\sin(r/2)|$, where $r = \theta - \theta_c$. Thus, on a periodic domain in 1D, we would use as a radial function, $\psi(r) = (x - x_c, y - y_c)$. Similarly in the case of a 2D doubly periodic domains, we use $\psi(x - x_c, y - y_c) = \phi(2\sqrt{\sin^2((x - x_c)/2) + \sin^2((y - y_c)/2)})$ [37].

8.4. Comparison of the RBF method with a finite difference scheme

The most commonly used methods to solve tsunami governing equations are finite difference schemes. The goal of this section is to show, by comparing RBFs with such a scheme, that RBFs, although more computationally expensive, have the potential to outperform the commonly used methods. The staggered leapfrog method (SL) is particularly well suited for this type of PDE (Equations (10)–(12)) and a simple geometry will allow for the grid staggering. It is a second-order method both in space and in time. For the sake of simplicity, instead of computing the analytic solution to the system, we choose a convenient solution to Equation (10) ($\eta(x, y) = \mathrm{e}^{-t/10} \sin(x)(-\sin(y) + \cos(y))$, $M(x, y) = \frac{1}{10}\mathrm{e}^{-t/10}\cos(x)\sin(y)$, and $N(x, y) = \frac{1}{10}\mathrm{e}^{-t/10}\sin(x)\sin(y)$), which creates forcing terms in Equations (11) and (12). This new system is the one that we solve numerically using the RBF and the SL schemes. The domain is a square equispaced grid, with N total nodes ($N^{1/2} \times N^{1/2}$ regular grid). Table II shows the relative max-norm error computed after a fixed time of $t = 5$, along with the minimum amount of grid points and the maximum possible time steps to reach it. We see that for a similar error, the method of lines with RBF used in the spatial

Table II. Comparison between RBF and the staggered leapfrog (SL) methods. We compare the smallest resolution and the largest time step allowed by each method to yield a similar error.

Method	Rel. l_∞ Error	N	Δt	Rel. l_∞ Error	N	Δt	Rel. l_∞ Error	N	Δt
RBF	9.34e−3	19^2	1.00	8.94e−4	26^2	0.55	1.01e−4	32^2	0.38
SL	1.08e−2	20^2	0.35	1.04e−3	65^2	0.11	1.05e−4	185^2	0.037

discretization and a 4th order Runge–Kutta scheme used in advancing the large set of ordinary differential equations associated with each grid point [35,36], (this is what we refer to when we mention the 'RBF method' in this section) allows for much larger time-steps and requires a much sparser grid than the leapfrog method. In realistic cases, using a method such as the SL method, we expect N to be around 650^2–1200^2. The asymptotic behaviors of the different curves allow to determine that, to obtain an equivalent error with the RBF method, we will only need N to be from 39^2 to 43^2 and the time steps will be from 24 to 41 times larger than the steps required when using the SL scheme. This observation is consistent with Flyer and Wright's conclusions in [34] and in [38] that although the RBF method has a higher complexity than most commonly used spectral methods, RBFs require a much lower resolution and a surprisingly larger time step than these methods. Overall, RBFs have therefore the potential to outperform these methods both in their complexity and in their accuracy. In addition, RBFs have the clear advantage of not being tied to any grid or coordinate system, making them suitable for difficult geometries. All this added to the code's undeniable simplicity makes the RBF method an excellent alternative to the commonly used methods, in particular in the context of tsunami modeling.

Figure 13 shows that the RBF method will shine over the finite difference method, as the number of grid points increases, because of its asymptotic property for larger time steps to be taken.

9. SUMMARY AND PERSPECTIVES FOR FUTURE

We have shown that the combination of GPU together with the use of RBFs makes good sense in terms of speeding up the tsunami wave computations. The linear shallow-water equations based on RBFs can be solved very fast on laptops, equipped with GPUs. They can be used at remote sites and can serve as beacons for warning the populace.

Modeling of tsunamis in the near-field close to the source may require 3-D formulation because of the recent findings about the potential importance of horizontal velocity field in the fluid movements [18]. For implementing the finite difference method in 3D, the method is also straightforward [15,22]. In 2D, we have used the 4-neighborhood connectivity for a simulation node. The data of neighboring simulation nodes is fetched via the texture fetching instructions. In 3D, we have to use the 18-neighborhood for a simulation node. The required data is fetched in the same way. This is a significantly more time-consuming sampling process, especially because of the fact that 3D texture sampling is much slower than 2D texture sampling. Therefore, we plan to adopt a simple acceleration approach, where a 3D texture volume is flattened into a large 2D texture. The 3D volume of texels is mapped to a 2D texture by laying out each of the $n \times n$ slices into a 2D texture

DOI: 10.1002/cpe

tile. At the same time, slice boundary should be carefully handled to avoid any sampling artifact. Since in 3D the simulation nodes become much more than in the 2D scenario, we expect that the speedup will be more significant.

We expect greater prospects from the improvements in double-precision on GPUs. There are two major brands in GPUs market: AMD (ATI) and NVIDIA. We have tested our method on the NVIDIA cards using CUDA. It is also possible to implement our method in AMD's platform. However, we need to put some major efforts to rewrite our code into AMD's GPU programming interface, Brook+. Initially, Brook is an extension of the C-language for GPU programming originally developed by Stanford University. AMD adopted Brook and extended it into Brook+ as the GPU programming specification on AMD's computation abstraction layer. In Brooks+, there are GPU data structures, which usually are called streams, and kernel function defined in the language extension. Streams are collections of data elements of the same type which can be operated on the GPU in parallel. Kernel functions are user-defined operations that operate on stream elements. The Brook+ source codes are compiled into C/C + + language codes via a customized preprocessor provided by AMD. Although functionally equivalent to the NVIDIA platform, AMD's platform has the advantage of supporting double precision computation as early as in late 2007. This is crucial in scientific computation where accuracy is often the first priority over speed. Without a proper level of accuracy, the simulation results will be useless. Recently, however, NVIDA has also announced the double precision support in its G200 series and up. It has been reported that the speed of double-precision computation is satisfactory [39] (a 16-fold speedup of double precision computation vs a 27-fold speedup of mixed computation of single/double-precision computation). Inspired by these results, we plan to pursue further the lores of double precision as long as the hardware is available to us. The recent introduction of the Tesla by NVIDIA, which is a third generation GPU dedicated for number crunching using 64-bit arithmetic, heralds a new era in desktop supercomputing, which will undoubtedly revolutionize the way tsunami warning will be issued in the future. Recently, Apply has introduced OpenCL, a new programming language that supports parallel execution on single or multiple processors whether they be CPUs or GPUs and is designed to work with OpenGL. In the next few years, OpenCL has the possibly of eclipsing CUDA. Thus, OpenCL allows for the possibility of further expanding the availability of supercomputing technologies, and may be another avenue that furthers the development of tsunami warnings in the future.

ACKNOWLEDGEMENTS

We thank Gordon Erlebacher, S. Mark Wang, Natasha Flyer, Tatsuhiko Saito, and Takahashi Furumura for helpful discussions. David A. Yuen also expresses support from the Earthquake Research Institute, University of Tokyo. This research has been supported by NSF grant to the Vlab at the Univ. of Minnesota. The National Center for Atmospheric Research is sponsored by the National Science Foundation. Cécile Piret was supported by NSF grant ATM-0620100 and the Advanced Studies Program at NCAR.

REFERENCES

1. Bryant E. *Tsunami*: *The Underrated Hazard* (2nd edn). Springer: Heidelberg, 2008; 330.
2. Levin B, Nosov M. *Physics of Tsunamis*. Springer: Heidelberg, 2009; 327.

DOI: 10.1002/cpe

3. Thuerey N, Muller-Fischer M, Schirm S, Gross M. Real-time breaking waves for shallow water simulations. *Proceedings of the Pacific Conference on Computer Graphics and Applications 2007*, IEEE Computer Society, October 2007; 8.
4. Thuerey N, Sadlo F, Schirm S, Muller-Fischer M, Gross M. Real-time simulations of bubbles and foam within a shallow water framework. *SCA '07*: *Proceedings of the 2007 ACM SIGGRAPH Eurographics Symposium on Computer Animation*, Eurographics Association, July 2007; 8.
5. Anderson JA, Lorenz CD, Travesset A. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics* 2008; **227**(10):5342–5539.
6. Nyland L, Harris M, Prins J. Fast N-body simulation with CUDA. *GPU Gems3*, Chapter 31. Addison-Wesley Professional: Reading, MA, 2007; 677–695.
7. Komatitsch D, Michea D, Erlebacher G. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *Journal of Parallel and Distributed Computing* 2009; **69**(5):451–460.
8. Walsh SDC, Saar MO, Bailey P, Liljia DJ. Accelerating geo-science and engineering system simulations on graphics hardware. *Computers and Geosciences* 2009; DOI: 10.1016/j.cageo.2009.05.001.
9. Fornberg B, Piret C. A stable algorithm for at radial basis functions on a sphere. *SIAM Journal on Scientific Computing* 2007; **200**:178–192.
10. Fasshauer GE. *Meshfree Approximation Methods with Matlab*. World Scientific Publishing: Singapore, 2007.
11. Ward SN. Tsunamis. In *The Encyclopedia of Physical Sciences and Technology* (3rd edn), Meyers RA (ed.), vol. 17. Academic Press: New York, 2002; 175–191.
12. Geist EL. Local tsunamis and earthquake source parameters. *Advances in Geophysics* 1997; **39**:117–209.
13. Geist EL, Dmowska R. Local tsunamis and distributed slip at the source. *Pure and Applied Geophysics* 1999; **154**:485–512.
14. Satake K. Tsunamis. *International Handbook of Earthquake and Engineering Seismology*, Lee WHK, Kanamori H, Jennings PC, Kisslinger C (eds.), vol. 81A. Elsevier Science & Technology Books, 2002; 437–451.
15. Gisler GR. Tsunami simulations. *Annual Review of Fluid Mechanics* 2008; **40**:71–90.
16. Imamura F, Gica E, Takahashi T, Shuto N. Numerical simulation of the 1992 Flores tsunami: Interpretation of tsunami phenomena in northeastern Flores Island and damage at Babi Island. *Pure and Applied Geophysics* 1995; **144**:555–568.
17. Okada Y. Surface deformation due to shear and tensile faults in a half-space. *Bulletin of the Seismological Society of America* 1985; **75**:1135–1154.
18. Song YT, Fu LL, Zlotnicki V, Ji C, Hjorleifsdottir V, Shum CK, Yi Y. The role of horizontal impulses of the faulting continental slope in generating the 26 December 2004 tsunami. *Ocean Modelling* 2008; **20**:362–379.
19. Shuto N, Goto C, Imamura F. Numerical simulations as a means of warning for near-field tsunamis. *Proceedings of the Second UJNR Tsunami Workshop*, Honolulu, Hawaii, 5–6 November 1990. National Geophysical Data Center, Boulder, 1991; 133–153.
20. Liu Y, Santos A, Wang SM, Shi Y, Liu H, Yuen DA. Tsunami hazards along the Chinese coast from potential earthquakes in South China sea. *Physics of the Earth and Planetary Interiors* 2007; **163**:233–244.
21. Kervella Y, Dutykh D, Dias F. Comparison between three-dimensional linear and nonlinear tsunami generation models. *Theoretical Computations in Fluid Dynamics* 2007; **21**:245–269.
22. Saito T, Furumura T. Three-dimensional simulation of tsunami generation and propagation: Application to intraplate events. *Journal of Geophysical Research* 2009; **114**. DOI: 10.1029/2007JB005523.
23. Mofjeld H, Symons C, Lonsdale P, Gonzalez F, Titov V. Tsunami scattering and earthquake faults in the deep Pacific Ocean. *Oceanography* 2003; **17**:38–46.
24. Sevre E, Yuen D, Liu Y. Visualization of tsunami waves with Amira package. *Visual Geosciences* 2008; **13**:85–96.
25. NVIDIA. CUDA compute unified device architecture. *Programming Guide*, *Version Beta 2.0*, 7 June 2008.
26. Kansa E. Multiquadrics—A scattered data approximation scheme with applications to computational fluid dynamics. II. Solutions to parabolic, hyperbolic and elliptic partial differential equations. *Computers and Mathematics with Applications* 1990; **19**:147–161.
27. Fornberg B, Zuev J. The Runge phenomenon and spatially variable shape parameters in RBF interpolation. *Computers and Mathematics with Applications* 2007; **54**:379–398.
28. Fornberg B, Flyer N. Accuracy of radial basis function interpolation and derivative approximations on 1-D infinite grids. *Advances in Computational Mathematics* 2005; **23**:5–20.
29. Fornberg B, Wright G. Stable computation of multiquadric interpolants for all values of the shape parameter. *Computers and Mathematics with Applications* 2004; **48**:853–867.
30. Fatica M, Jeong W. Accelerating MATLAB with CUDA. *Proceedings of the Eleventh Annual High Performance Embedded Computing Workshop*, Lexington, Massachusetts, 18–20 September 2007.
31. Melonakos J. Parallel computing on a personal computer. *Biomedical Computation Review* 2008; **4**:29.
32. AccelerEyes. Jacket user guide: MATLAB GPU programming. *Programming Guide*, *Version Beta 0.5*, 19 September 2008.
33. Weiskopf D. *GPU Based Interactive Visualization Techniques* (1st edn). Springer: New York, 2006; 312.
34. Flyer N, Wright G. Transport schemes on a sphere with radial basis functions. *Journal of Computational Physics* 2007; **226**:1059–1084.

35. Schiesser WE. *The Numerical Method of Lines*. Academic Press: San Diego, 1991.
36. Dahlquist G, Bjoerck A. *Numerical Methods in Scientific Computing*, vol. 1. SIAM: Philadelphia, 2008.
37. Fornberg B, Flyer N, Russell J. Comparisons betpseudospectral and radial basis function derivative approximations. *IMA Journal of Numerical Analysis* 2008; DOI: 10.1093/imanum/dri000.
38. Flyer N, Wright G. A radial basis function shallow water model. *Proceedings of the Royal Society of London*, *Series A* 2009; **465**(2106):1949–1976.
39. Göddeke D, Strzodka R. *Performance and Accuracy of Hardware-oriented Native-*, *Emulated- and Mixed-precision Solvers in FEM Simulations* (*Part 2*: *Double Precision GPUs*), Ergebnisberichte des Instituts für Angewandte Mathematik, Nr. 370, TU Dortmun, 2008.

DOI: 10.1002/cpe